CSCI 280 Fall 2012 **Greedy Exchange**

Greedy solutions generally take the following form. Select a candidate greedily according to some heuristic, and add it to your current solution if doing so doesn't corrupt feasibility. Repeat if not finished. "Greedy Exchange" is one of the techniques used in proving the correctness of greedy algorithms. The idea of a greedy exchange proof is to incrementally modify a solution produced by any other algorithm into the solution produced by your greedy algorithm in a way that doesn't worsen the solution's quality. Thus the quality of your solution is at least as great as that of any other solution. In particular, it is at least as great as an optimal solution, and thus, your algorithm does in fact return an optimal solution.

Main Steps

After describing your algorithm, there are three main steps for a greedy exchange argument proof.

Step 1: Label your algorithm's solution, and a general solution. For example, let $A = \{a_1, a_2, \ldots, a_k\}$ be the solution generated by your algorithm, and let $O = \{o_1, o_2, \ldots, o_m\}$ be an arbitrary (or optimal) feasible solution.

Step 2: Compare greedy with other solution. Assume that your arbitrary/optimal solution is not the same as your greedy solution (since otherwise, you are done). Typically, you can isolate a simple example of this difference, such as one of the following.

- there is an element of O that is not in A and an element of A that is not in O, or
- there are 2 consecutive elements in O in a different order than they are in A (i.e. there is an inversion).

Step 3: Exchange. Swap the elements in question in O (either swap one element out and another in for the first case, or swap the order of the elements in the second case), and argue that you have a solution that is no worse than before. Then argue that if you continue swapping, you can eliminate all differences between O and A in a finite number of steps without worsening the quality of the solution. Thus, the greedy solution produced is just as good as any optimal (or arbitrary) solution, and hence is optimal itself.

Comments

- Remember you need to argue that *multiple* swaps can get you from your selected solution to greedy, as one single swap will usually not suffice. Also, make sure that any step you make (and not just the first one) doesn't hurt the solution quality.
- Note that the swapping is not part of your algorithm; it is only part of the argument that shows your algorithm is correct.

Example: Minimum Spanning Tree

We are given a graph $G = (V, E)$, with costs on the edges, and we want to find a spanning tree of minimum cost. We use Kruskal's algorithm, which sorts the edges in order of increasing cost, and tries to add them in that order, leaving edges out only if they create a cycle with the previously selected edges.

Proof of Correctness for Kruskal's Algorithm: Let $T = (V, F)$ be the spanning tree produced by Kruskal's algorithm, and let $T^* = (V, F^*)$ be a minimum spanning tree. If $F = F^*$ then T is optimal and we are done; otherwise $F^* \neq F$, and there is an edge $e \in F^*$ such that $e \notin F$. Then e creates a cycle C in the graph $T + \{e\}$, and at least one edge f of this cycle crosses the cut defined by $T^* - \{e\}$. Furthermore, the reason e is not in F must be that when the algorithm considered adding e, the rest of C was already in the tree. Since we consider edges in increasing order of cost, this means that e must be the most expensive edge in C, and so $cost(f) \leq cost(e)$. If we add the edge f to the graph $T^* - \{e\}$, then we reconnect the graph and create a spanning tree.

Also, $cost(T^* - \{e\} + \{f\}) = cost(T^*) - cost(e) + cost(f) \leq cost(T^*)$, and so we have created a new spanning tree of no more cost than T^*, but with one more edge in common with T. We can do this for every edge that differs between T and T^*. The two trees differ on at most all $n - 1$ edges, so after at most $n - 1$ steps we obtain the tree T of no more cost than T^*. This contradicts the assumption that T was not optimal.

The algorithm begins by sorting the m edges in order of increasing cost; this take time $O(m \log m)$. Since $m \leq n^2$, this is also $O(m \log n)$. We then use the union-find data structure to keep track of the components of our greedy solutions; for each edge we consider, we perform two `find` operations, and possibly one `union` operation. That is, in the worst case, we perform $2m$ `union`s and $n - 1$ `union`s; this part of the algorithm takes time $O(m \log n)$. Therefore, the total running time is $O(m \log n)$.